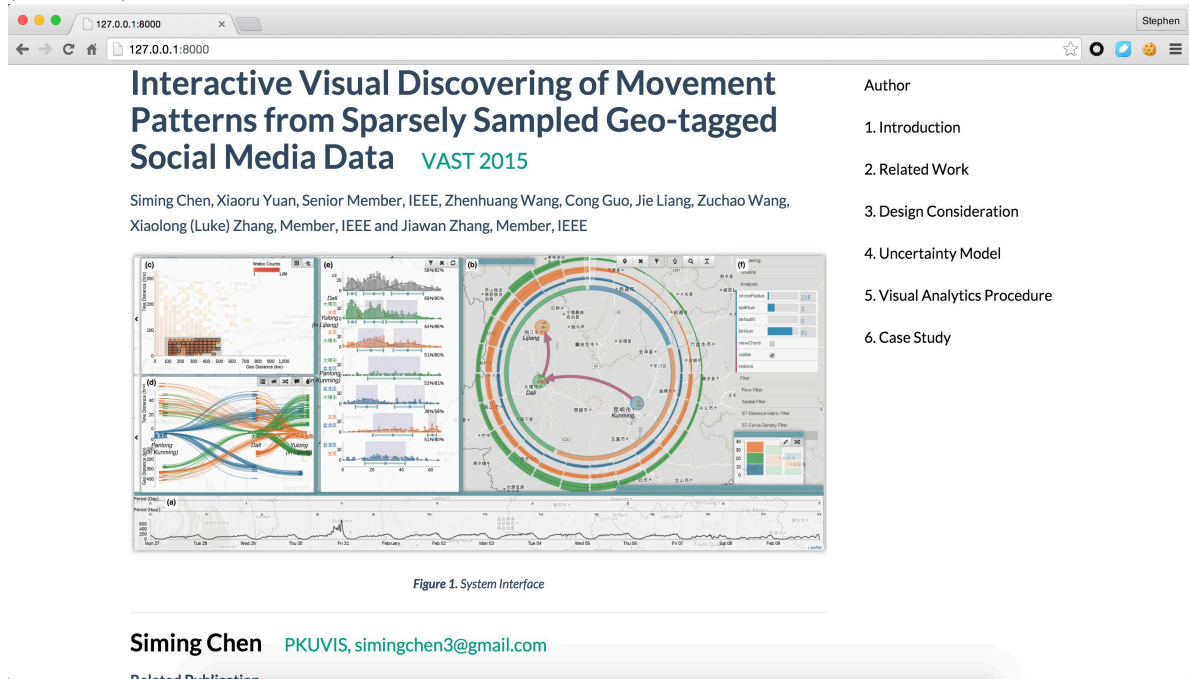


1. 本周工作

1.1 组会报告

研读了Interactive Visual Discovering of Movement Patterns from Sparsely Sampled Geo-tagged Social Media Data (VAST-2015), 花了比较多的时间, 制作了一份HTML, 用于组会报告:



1.2 拜读张繁老师的论文《屏幕拼接可视化技术的研究进展》

1.3 研读《Modern Operating Systems》(4th Edition) -- Chapter 2. Process and Thread

本科是读的是《Operating System Concept》(后称concept), 最近在知乎上看到别人推荐这本书, 便阅读了与大屏项目相关的进程、线程部分。本章相比《Concept》一书, 在介绍Process(进程)、Thread(线程)、Interprocess Communication(进程间通信, IPC)的基础概念的同时, 在IPC部分, 提出了更多的模型。

第二章由许多部分组成, 对于IPC部分, 需要明确和值得一提的概念、模型有:

1.3.1 Race Conditions (竞态)

当两个或多个进程访问(读、写)共享的数据时, 若程序结果受这它们的访问顺序影响, 此时将出现静态。

1.3.2 Critical Regions (临界区)

通过引入临界区来避免竞态的产生。但为了使同步进程比较有效率地运行, 还需要保持下述四个条件:

- 同时不能有两个进程都在临界区中
- 不能对CPU的数量和速度予以假设
- 在临界区外的进程不可以阻塞其他进程
- 想要进入临界区的进程不可以永久等待

1.3.3 Mutual Exclusion with Busy Waiting (忙等待的互斥)

```

// Process 0
while (true)
{
    while (turn != 0) ;
    critical_region();
    turn = 1;
    noncritical_region();
}

// Process 1
while (true)
{
    while (turn != 1) ;
    critical_region();
    turn = 0;
    noncritical_region();
}

```

可以通过锁turn来决定哪个进程进入临界区。

这是我今年3月份，在斐然师兄的那篇paper里，采用的方法。但当时因为没有将while (turn != 0)这种锁的检测写成原子性操作，导致同步的失败。

1.3.4 Sleep and Wakeup (挂起与唤醒)

为解决“忙等待问题”，引入了“挂起与唤醒”的概念。介绍了经典的Producer-Consumer Problem (bounded-buffer problem).

通过访问共享变量count，进程可以：

- 主动交出CPU(sleep)
- 继续执行，选择性的唤醒别的进程(wakeup) -- 让别的进程处于waiting状态。

1.3.5 Semaphores (信号量)

1.3.4提到的共享变量count是没有经过同步的 —— 存在两个进程同时读写这个变量的可能。

为解决这个问题，引入了信号量的概念。通过原子性的P(down), V(up)操作来维护信号量：

- 进程调用P(semaphores)时，考察信号量是否大于0
 - 若是，信号量减一，进程继续运行
 - 反之，信号量保持0不变，进程挂起在这个信号量上。
- 进程调用V(semaphores)操作，考察是否有进程挂起在这个信号量上
 - 若无，信号量增一，本进程继续运行
 - 若有，将其中一个挂起进程从block状态转换成waiting转换，信号量保持不变，本进程继续进行

注意到，互斥锁，可以看做是初值为1的二元信号量(binary semaphore).

1.3.6 Mutexes (互斥锁)

亦即是线程安全的共享变量。

注意到，当程序有多个互斥锁锁，每个进程中，加解锁的顺序应该是相同的，否则可能出现循环等待，导致死锁。

1.3.7 Monitor (监听者)

没有读这一部分：

- C, C++语言特性不支持这种模型；
- Java支持。

1.3.8 Message Passing (消息传递)

比较像是进程在监听某一个端口：

- 若没有消息到达，则挂起；
- 若消息到达，则执行或被唤醒。

1.3.9 Barrier (栅栏)

用于多进程的运行边界同步，假设：

- P1, P2, P3... 要执行阶段一和阶段二的任务，但阶段二的任务必须
- 在任一进程未完成阶段一的任务时，所有进程都不得执行阶段二

这时就需要引入栅栏机制，保证所有进程都被挡在阶段二之前。

这是我在利用多线程进行力引导算法的迭代中用的机制 —— 将引力、斥力的计算分发到各个线程中，将每次迭代看做一个阶段。这种方法将4核(亦即8虚拟CPU)PC机的迭代速度提升了4倍。

2. 下周工作

研读姜老师的开放课题申请书；

研读《C++ Concurrency in action》；

在老师的指导下展开调研工作。